Manual de Scala
para programadores Java

Version 1.2
July 12, 2007

Michel Schinz,
Philipp Haller,
Juanjo Bazán (ES ver.)

PROGRAMMING METHODS LABORATORY
EPFL
SWITZERLAND

1 Introducción

Este manual proporciona una introducción rápida al lenguaje Scala y a su compilador. Está dirigido a personas que ya tengan algo de experiencia programando y deseen echar un vistazo a las posibilidades de Scala. Se presuponen conocimientos básicos de programación orientada a objetos, especialmente en Java.

2 Un primer ejemplo

Como primer ejemplo, utilizaremos el habitual programa *Hola Mundo*. No es muy fascinante pero es una manera fácil de mostrar el uso de las herramientas Scala sin que sea necesario conocer mucho sobre el lenguaje. Esta es la pinta que tiene:

```
object HolaMundo {
  def main(args: Array[String]) {
    println("Hola, mundo!")
  }
}
```

La estructura de este programa debería ser familiar para cualquier programador Java: consiste en un método llamado `main` que toma como parámetro de entrada en forma de array de cadenas de texto(Strings) los argumentos introducidos en la línea de comandos. El cuerpo de este método consiste en una única llamada al método predefinido `println` con una saludo amistoso como argumento. El método `main` no devuelve ningún valor, por tanto no es necesario declarar ningún tipo de respuesta.

Lo que es menos familiar para los programadores Java es la declaración de **`object`** que contiene al método `main`. Esta declaración nos presenta lo que se conoce comunmente como *objeto singleton*, es decir, una clase con una única instancia. Por tanto, la declaración anterior declara tanto una clase llamada `HolaMundo` como una instancia de esa clase, también llamada `HolaMundo`. Esta instancia se crea bajo demanda, la primera vez que se utiliza.

El lector astuto se habrá dado cuenta de que el método `main` no está declarado como `static`. Esto se debe a que los miembros estáticos(métodos o variables) no existen en Scala. En vez de definir miembros estáticos, el programador Scala declara esos métodos como objetos singleton.

2.1 Compilando el ejemplo

Para compilar el ejemplo utilizamos `scalac`, el compilador de Scala. `scalac` funciona como la mayoría de los compiladores: recibe una archivo de código fuente como argumento, quizá algunas opciones, y produce uno o varios archivos. Los archivos de objetos que produce son archivos estándar de clases Java.

Si guardamos el programa anterior en un fichero llamado `HolaMundo.scala`, pode-

mos compilarlo mediante la siguiente instrucción(el símbolo mayor que '>' representa el indicador de la shell y no hace falta escribirlo):

```
> scalac HolaMundo.scala
```

Esto genera unas cuantos archivos de clases en el directorio actual. Uno de ellos se llamará `HolaMundo.class`, y contendrá una clase que puede ser ejecutada directamente mediante el comando `scala`, como se muestra en la siguiente sección.

2.2 Ejecutando el ejemplo

Una vez compilado, un programa Scala se puede ejecutar utilizando el commando `scala`. Su uso es muy similar al del comando `java` utilizado para ejecutar programas Java, y acepta las mismas opciones. El ejemplo anterior puede ejecutarse mediante el siguiente comando, que produce la salida esperada:

```
> scala -classpath . HolaMundo

Hola, mundo!
```

3 Interacción con Java

Uno de los puntos fuertes de Scala es que permite muy facilmente interactuar con código Java. Por defecto se importan todas las clases del paquete `java.lang`, el resto necesitan ser importadas de manera explícita.

Veamos un ejemplo que nos lo muestre. Queremos obtener la fecha actual y darle formato de acuerdo a la convención de un país específico, digamos Francia[1].

Las librerias de Java tienen definidas clases muy potentes como `Date` y `DateFormat`. Puesto que Scala interopera directamente con Java, no hay necesidad de implementar clases equivalentes en la libreria de clases de Scala, basta simplemente con importar las clases de los correspondientes paquetes de Java:

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

object FrenchDate {
  def main(args: Array[String]) {
    val now = new Date
    val df = getDateInstance(LONG, Locale.FRANCE)
    println(df format now)
  }
}
```

[1] la misma convención se aplica en otras regiones como la zona francófona de Suiza.

La instrucción para importar en Scala se parece mucho a su equivalente de Java, pero en realidad es más potente. Se pueden importar varias clases del mismo paquete encerrandolas entre llaves como en la primera línea. Otra diferencia es que cuando queremos importar todos los nombres de un paquete o de una clase, se utiliza el carácter guión bajo (_) en vez del asterisco (`*`). Esto se debe a que el asterisco es un identificador válido en Scala (es decir, el nombre de un método), como veremos más adelante.

Por tanto el import de la tercera línea importa todos los miembros de la clase `DateFormat`. Esto hace que sean visibles directamente el método estático `getDateInstance` y el campo estático `LONG`.

Dentro del método `main` primero creamos una instancia de la clase `Date` de Java que por defecto contiene la fecha actual. Después definimos un formato de fecha utilizando el método estático `getDateInstance` que hemos importado previamente. Y por último, mostramos la fecha actual formateada de acuerdo a la instancia localizada de `DateFormat`. Los métodos que reciben un solo argumento se pueden usar con sintaxix infix, es decir, la expresión

```
df format now
```

es simplemente otra manera ligeramente más corta de escribir la expresión

```
df.format(now)
```

Esto puede parecer un detalle sintáctico menor, pero tiene consecuencias importantes, una de las cuales detallaremos en la próxima sección.

Para acabar con este apartado sobre la interacción con Java, hay que mencionar que en Scala es también posible heredar de clases Java e implementar interfaces de Java.

4 Todo es un objeto

Scala es un lenguaje orientado a objetos puro, en el sentido de que *todo* es un objeto, incluidos los números o las funciones. En ese aspecto se diferencia de Java, puesto que Java distingue tipos primitivos(como `int` y `boolean`) y tipos por referencia, y no permite manipular funciones como valores.

4.1 Los números son objetos

Puesto que los números son objetos, también tienen métodos. Y de hecho una expresión aritmética como esta:

```
1 + 2 * 3 / x
```

esta formada exclusivamente de llamadas a métodos, porque es equivalente a la siguiente expresión, como vimos en la sección anterior:

```
1.+(2.*(3./(x)))
```

Lo que significa también que `+`, `*`, etc. son identificadores válidos en Scala.

4.2 Las funciones son objetos

Quizá más sorprendente para el programador Java, las funciones también son objetos en Scala. Por tanto es posible pasar funciones como argumentos, guardarlas en variables y devolverlas como respuesta de otras funciones. Esta habilidad para manipular funciones como valores es una de las piedras angulares de un interesante paradigma de programación llamado *programación funcional*.

Como ejemplo muy simple de porqué puede resultar útil el uso de funciones como valores, consideremos una función temporizador cuyo propósito sea realizar alguna acción una vez por segundo. ¿Cómo le pasamos la acción a realizar? lógicamente, como una función. Esta manera sencilla de pasar funciones debería ser familiar para muchos programadores: se utiliza frecuentemente en código de interfaces de usuario, para llamadas a funciones que hay que realizar cuando se produce algún evento.

En el siguiente programa, la función temporizador se llama `unaVezPorSegundo`, y toma una función como argumento. El tipo de esta función se escribe `() => unit` y es el tipo de todas las funciones que no tienen parámetros de entrada y que no devuelven nada(el tipo `unit` es similar a `void` en C/C++). El método main de este programa simplemente llama a la función temporizador para que repita cada segundo la impresión por pantalla de la frase “el tiempo pasa volando”.

```
object Temporizador {
  def unaVezPorSegundo(repite: () => unit) {
    while (true) { repite(); Thread sleep 1000 }
  }
  def elTiempoVuela() {
    println("el tiempo pasa volando...")
  }
  def main(args: Array[String]) {
    unaVezPorSegundo(elTiempoVuela)
  }
}
```

Notese como para imprimir la cadena de texto usamos el método predefinido `println` en vez de usar el método de `System.out`.

4.2.1 Funciones anónimas

Aunque este programa es fácil de entender, se puede refinar un poco. Antes de nada, se ve que la función `elTiempoVuela` solo se ha creado para ser pasada posterior-

mente como argumento al método `unaVezPorSegundo`. Tener que dar nombre a esa función, que sólo se usa una vez, parece innecesario, y de hecho estaría bien poder construirla justo cuando se le pasa a `unaVezPorSegundo`. En Scala esto es posible usando *funciones anónimas*, que son exactamente eso: funciones sin nombre. La versión revisada de nuestro programa temporizador utilizando una función anónima en vez de `elTiempoVuela` queda así:

```
object Temporizador {
  def unaVezPorSegundo(repite: () => unit) {
    while (true) { repite(); Thread sleep 1000 }
  }
  def main(args: Array[String]) {
    unaVezPorSegundo(() =>
      println("el tiempo pasa volando..."))
  }
}
```

La presencia de una función anónima en este ejemplo, se revela por la flecha '=>' que separa a la lista de argumentos de la función, del cuerpo de la misma. En este ejemplo la lista de argumentos es vacía, como indican los paréntesis vacíos a la izquierda de la flecha. El cuerpo de la función es el mismo que el de `elTiempoVuela` antes.

5 Clases

Como hemos visto, Scala es un lenguaje orientado a objetos, y como tal tiene el concepto de clase.[2] Las clases en Scala se declaran usando una sintaxis próxima a la de Java. Una diferencia importante es que las clases en Scala pueden tener parámetros. Lo ilustramos con la siguiente definición de números complejos:

```
class Complex(real: double, imaginary: double) {
  def re() = real
  def im() = imaginary
}
```

Esta clase recibe dos argumentos, que son la parte real y la parte imaginaria del número complejo. Estos argumentos hay que pasarlos cuando se crea una instancia de la clase `Complex`, de la siguiente manera: `new Complex(1.5, 2.3)`. La clase contiene dos métodos, llamados `re` e `im`, que permiten el acceso esas dos partes.

Hay que hacer notar que el tipo de respuesta de esos dos métodos no está dado explicitamente. El compilador lo inferirá automáticamente mirando al lado derecho de los dos métodos para deducir que ambos devuelven un valor de tipo `double`.

[2] Por completitud, hay que decir que algunos lenguajes orientados a objetos no tienen el concepto de clase, pero Scala no es uno de ellos.

El compilador no es capaz de deducir tipos siempre como hace aquí, y desafortunadamente no hay una regla simple para saber cuando lo será y cuando no. En la práctica generalmente esto no es un problema puesto que el compilador se queja cuando no es capaz de inferir un tipo que no se explicita. Como regla simple, los programadores principiantes de Scala deberian omitir las declaraciones de tipos que parezcan fáciles de deducir a partir del contexto, y ver si el compilador está de acuerdo. Después de un tiempo, el programador habrá desarrollado un buen criterio sobre cuando omitir los tipos y cuando declararlos explicitamente.

5.1 Métodos sin argumentos

Un pequeño problema con los métodos `re` e `im` es que, para poder llamarlos, uno ha de poner un par de paréntesis después del nombre, como muestra el siguiente ejemplo:

```
object NumerosComplejos {
  def main(args: Array[String]) {
    val c = new Complex(1.2, 3.4)
    println("parte imaginaria: " + c.im())
  }
}
```

Estaría bien poder acceder a las partes real e imaginaria como si fueran variables, sin tener que poner los paréntesis. Esto se puede hacer sin problemas en Scala, simplemente definiendolos como métodos *sin argumentos*. Tales métodos difieren de los métodos con cero argumentos en que no llevan paréntesis tras su nombre, ni en su definición ni en su uso. Nuestra clase `Complex` se puede reescribir así:

```
class Complex(real: double, imaginary: double) {
  def re = real
  def im = imaginary
}
```

5.2 Herencia y sobreescritura

Todas las clases en Scala heredan de una super-clase. Cuando no se especifica superclase, como en el ejemplo de la clase `Complex` de la sección previa, implicitamente se usa `scala.Object`.

En Scala es posible sobreescribir métodos heredados de una superclase. Pero es obligatorio especificar explícitamente que un método sobreescribe a otro usando el modificador **`override`**, para evitar sobreescrituras accidentales. Como ejemplo podemos aumentar nuestra clase `Complex` con la redefinición del método `toString` heredado de `Object`.

```
class Complex(real: double, imaginary: double) {
```

```
  def re = real
  def im = imaginary
  override def toString() =
    "" + re + (if (im < 0) "" else "+") + im + "i"
}
```

6 Clases Case y comparación por patrones

Un tipo de estructura frecuente en aplicaciones es la de árbol. Por ejemplo, los intérpretes y los compiladores normalmente representan internamente a los programas como árboles; los documentos XML son árboles; y muchos tipos de contenedores están basados en árboles, como por ejemplo los árboles rojo/negro.

Examinaremos ahora como representar y manipular estos árboles es Scala a través de un pequeño programa calculadora. El propósito del programa es manipular expresiones aritméticas muy simples formadas por sumas, enteros y variables. Dos ejemplos de tales expresiones son $1+2$ y $(x+x)+(7+y)$.

Primero tenemos que decidir una manera de representar este tipo de expresiones. LA forma más natural es un árbol, cuyos nodos son operaciones(en este caso la suma) y cuyas hojas son valores(aquí constantes o variables).

En Java, se podría representar este árbol usando una superclase abstracta para los árboles y una subclase concreta para nodo u hoja. En un lenguaje de programación funcional, uno podría usar para el mismo propósito un tipo de dato algebraico. Scala proporciona el concepto de *clases case* que de alguna manera es algo intermedio entre los dos. Aquí está como se pueden usar para definir el tipo de los árboles de nuestro ejemplo:

```
abstract class Arbol
case class Sum(l: Arbol, r: Arbol) extends Arbol
case class Var(n: String) extends Arbol
case class Const(v: int) extends Arbol
```

El hecho de que las clases `Sum`, `Var` y `Const` sean declaradas como clases case significa que son diferentes de las clases estándar en varios aspectos:

- la palabra clave **`new`** no es obligatoria para crear instancias de estas clases (es decir, podemos escribir `Const(5)` en vez de **`new`** `Const(5)`),
- automáticamente se definen funciones de acceso a los parámetros del constructor(es decir, es posible obtener el valor del parámetro del contructor `v` de una instancia `c` de la clase `Const` simplemente escribiendo `c.v`),
- se proporciona definición por defecto para los métodos `equals` y `hashCode`, que funcionan sobre la *estructura* de las instancias y no sobre su identidad,

- se proporciona una definición por defecto para el método `toString`, que escribe el valor en "formato fuente (source form)" (es decir, el árbol de la expresión $x+1$ se escribe como `Sum(Var(x),Const(1))`),
- se pueden diferenciar instancias de estas clases mediante *concordancia con patrones(pattern matching)* como veremos posteriormente.

Una vez que hemos definido el tipo de datos con el que representar nuestras expresiones aritméticas podemos comenzar a definir operaciones con las que manipularlas. Empezaremos con una función para evaluar una expresión dentro de un *entorno*. El propósito del entorno es el de proporcionar valores a las variables. Por ejemplo, la expresión $x+1$ evaluada en un entorno que asocia el valor 5 a la variable x, escrito $\{x \rightarrow 5\}$, da como resultado 6.

Por tanto tenemos que encontrar una manera de representar entornos. Podríamos hacerlo claro está usando algún tipo de estructura de datos asociativa como una tabla hash, pero también podemos ¡usar funciones directamente!. En realidad un entorno no es más que una función que asocia un valor a un nombre(de variable). El entorno $\{x \rightarrow 5\}$ de antes puede ser escrito fácilmente en Scala así:

```
{ case "x" => 5 }
```

Está notación define una función que, cuando recibe la cadena de texto `"x"` como argumento, devuelve el entero 5, en cualquier otro caso falla con una excepción.

Antes de escribir la función de evaluación, demos un nombre al tipo de datos de los entornos. Podríamos usar siempre el tipo `String => int` para los entornos, pero nuestro programa se simplifica si introducimos un nombre para este tipo de dato, y además así los posibles cambios futuros serán más fáciles de realizar. Esto se consigue en Scala usando la siguiente notación:

```
type Entorno = String => int
```

A partir de ahora, el tipo `Entorno` se puede usar como un alias del tipo de funciones de `String` a `int`.

Podemos ahora dar la definición de la función de evaluación. Conceptualmente es muy simple: el valor de la suma de dos expresiones es simplemente la suma del valor de esas expresiones; el valor de una variable se obtiene directamente del entorno; y el valor de una constante es la propia constante. Expresar esto en Scala no es mucho más dificil:

```
def eval(t: Arbol, ent: Entorno): int = t match {
  case Sum(l, r) => eval(l, ent) + eval(r, ent)
  case Var(n)    => env(n)
  case Const(v)  => v
}
```

Esta evaluación funciona realizando *comparación de patrones(pattern matching)* sobre el árbol `t`. Intuitivamente el significado de la definición anterior debería quedar claro:

1. primero se comprueba si el árbol `t` es de tipo `Sum`, y si lo es, asigna el sub-árbol izquierdo a una variable llamada `l`, el sub-árbol derecho a una variable llamada `r` y luego procede a evaluar la expresión al otro lado de la flecha; esta expresión puede(y lo hace) hacer uso de la asignación de variables hecha en el patrón que aparece a la izquierda de la flecha, es decir `l` y `r`,
2. si la primera comprobación no se cumple, es decir el árbol no es un `Sum`, continúa y comprueba si `t` es de tipo `Var`; si lo es, asigna el nombre que contiene el nodo `Var` a la variable `n` y pasa a la expresión de la derecha,
3. si la segunda comprobación también falla, es decir si `t` no es ni `Sum` ni `Var`, comprueba si es `Const`, y si lo es, asigna el valor que haya en el nodo `Const` a la variable `v` y prosigue con el lado derecho,
4. por último, si todas las comprobaciones fallan, se lanza una excepción para señalar el fallo en la concordancia de la expresión con los patrones con los que se compara. Lo que aquí sólo podría pasar si se declararan más sub-clases de `Arbol`.

Vemos que la idea básica de la concordancia con patrones es intentar comparar un valor con una serie de patrones, y en cuanto un patrón encaja, extraer y nombrar varias partes del valor, para finalmente evaluar código que típicamente utiliza esas partes recién nombradas.

Un programador curtido en orientación a objetos podría preguntarse porque no hemos definido `eval` como un *método* de la clase `Arbol` y de sus subclases. En realidad se podría haber hecho así puesto que Scala permite definir métodos en clases-case igual que en cualquier otra clase normal. Decidir cuando usar patrones o cuando usar métodos es en realidad una cuestión de gustos, pero también hay implicaciones importantes sobre extensibilidad:

- cuando se usan métodos, es fácil añadir un nuevo tipo de nodo puesto que basta con definir la subclase de `Arbol` correspondiente; pero por otro lado, añadir una operación nueva para manipular el arbol es tedioso puesto que requiere la modificación de todas las subclases de `Arbol`,
- cuando se usa concordancia de patrones la situación se invierte: añadir un nuevo tipo de nodo requiere modificar todas las operaciones que comparan patrones contra el árbol para que tengan en cuenta el nuevo tipo de nodo; por otro lado, añadir una nueva operación es fácil, definiendola simplemente como una función independiente.

Para explorar en mayor profundidad la concordancia con patrones, definamos otra operación sobre operaciones aritméticas: la derivada simbólica. El lector seguramente recordará las reglas relativas a esta operación:

1. la derivada de una suma es la suma de sus derivadas,
2. la derivada de una variable v es uno si v es la variable respecto a la que se deriva, y cero en otro caso,
3. la derivada de una constante es cero.

Estas reglas se pueden traducir a código Scala casi literalmente, para obtener la siguiente definición:

```
def deriva(t: Arbol, v: String): Arbol = t match {
  case Sum(l, r) => Sum(deriva(l, v), deriva(r, v))
  case Var(n) if (v == n) => Const(1)
  case _ => Const(0)
}
```

Esta función introduce dos nuevos conceptos relacionados con la comparación con patrones. En primer lugar la expresión **case** para las variables tiene un requisito, una expresión seguida de la palabra clave **if**. Este requisito previene que la concordancia con el patrón tenga éxito a menos que la expresión se cumpla. Aquí se usa para asegurarnos de que devolvemos la constante 1 sólo si el nombre de la variable que estamos derivando es el mismo que la variable de la derivada, v. La segunda funcionalidad nueva de la comparación con patrones que usamos aquí es el *comodín(o wild-card)*, escrito como _, que es un patrón que concuerda con cualquier valor, sin tener que darle un nombre.

Aún no hemos explorado toda la potencia que nos proporciona la comparación con patrones, pero lo dejamos en este punto para mantener la brevedad de este documento. Pero todavia queremos ver como funcionan las dos funciones anteriores con un ejemplo real. Para este propósito, escribamos una función `main` simple que realice varias operaciones sobre la expresión $(x+x)+(7+y)$: Primero calcula su valor en el entorno $\{x \to 5, y \to 7\}$, y luego calcula sus derivadas respecto a x y respecto a y.

```
def main(args: Array[String]) {
  val exp: Arbol = Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")))
  val ent: Entorno = { case "x" => 5 case "y" => 7 }
  println("Expresión: " + exp)
  println("Evaluamos siendo x=5, y=7: " + eval(exp, ent))
  println("Derivada respecto a x:\n " + deriva(exp, "x"))
  println("Derivada respecto a y:\n " + deriva(exp, "y"))
}
```

Al ejecutar este programa se obtiene la salida esperada:

```
Expresión: Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y)))
Evaluamos siendo x=5, y=7: 24
Derivada respecto a x:
```

```
 Sum(Sum(Const(1),Const(1)),Sum(Const(0),Const(0)))
Derivada respecto a y:
 Sum(Sum(Const(0),Const(0)),Sum(Const(0),Const(1)))
```

Si examinamos la salida del programa, vemos que el resultado de la derivada debería simplificarse antes de mostrarlo al usuario. Definir una función de simplificación básica usando comparación con patrones es un problema interesante(aunque sorprendentemente complicado), que se deja como ejercicio al lector.

7 Mixins

Aparte de heredar código de una super-clase, una clase de Scala puede también importar código de uno o varios *mixins*. Para un programador Java quizá la manera más fácil de entender lo que son los mixins es verlos como interfaces que también pueden contener código. En Scala, cuando una clase hereda de un mixin, implementa el interfaz de ese mixin, y hereda todo el código contenido en el mixin.

Para ver la utilidad de los mixins, echemos un vistazo a un ejemplo clásico: objetos ordenados. A menudo es útil poder comparar objetos de una clase dada entre ellos, por ejemplo para ordenarlos. En Java, los objetos que que son comparables implementan el interfaz `Comparable`. En Scala, podemos hacerlo algo mejor que en Java definiendo nuestro equivalente a `Comparable` como un mixin. que llamaremos `Ord`.

Cuando se comparan objetos pueden ser útiles seis predicados diferentes: menor, menor o igual, igual, no igual, mayor o igual y mayor. Pero definir todos ellos es fastidioso, especialmente cuando cuatro de esos seis se pueden expresar usando los otros dos. Es decir dados los predicados igual y menor(por ejemplo), uno puede expresar los demás. En Scala todas estas observaciones se pueden plasmar de manera sencilla declarando el siguiente mixin(mediante la palabra clave **`trait`**):

```
trait Ord {
  def < (that: Any): boolean
  def <=(that: Any): boolean = (this < that) || (this == that)
  def > (that: Any): boolean = !(this <= that)
  def >=(that: Any): boolean = !(this < that)
}
```

Esta definición crea tanto un nuevo tipo llamado `Ord`, que juega el mismo papel que el interfaz `Comparable` de Java, como implementación por defecto para tres predicados en términos de un cuarto, abstracto. Los predicados para la igualdad y la desigualdad no aparecen aquí puesto que por defecto están presentes en todos los objetos.

El tipo de dato `Any` usado en el ejemplo anterior es el tipo que es super-tipo de todos los demás tipos en Scala. Puede verse como una versión más general del tipo de Java `Object`, puesto que también es un super-tipo de los tipos básicos como `int`, `float`,

etc.

Para hacer comparables los objetos de una clase, es por tanto suficiente con definir los predicados que comprueban la igualdad y la inferioridad, y añadir la clase `Ord` anterior. Como ejemplo, vamos a definir una clase `Date` que represente fechas en el calendario Gregoriano. Estas fechas se componen de un día, un mes y un año, que representaremos en los tres casos como enteros. Así pues empezamos con la definición de la clase `Date` como sigue:

```
class Date(y: int, m: int, d: int) extends Ord {
  def year = y
  def month = m
  def day = d

  override def toString(): String = year + "-" + month + "-" + day
```

La parte importante aquí es la declaración **`extends`** `Ord` que sigue al nombre y a los parámetros de la clase. Declara que la clase `Date` hereda de la clase `Ord` como un mixin.

Ahora redefinimos el método `equals`, heredado de `Object`, para que compare correctamente fechas comparando individualmente sus campos. La implementación por defecto de `equals` no es válida porque al igual que en Java compara los objetos físicamente. Obtenemos la siguiente definición:

```
  override def equals(that: Any): boolean =
    that.isInstanceOf[Date] && {
      val o = that.asInstanceOf[Date]
      o.day == day && o.month == month && o.year == year
    }
```

Este método hace uso de los métodos predefinidos `isInstanceOf` y `asInstanceOf`. El primero de ellos, `isInstanceOf`, corresponde al operador `instanceof` de Java, y devuelve verdadero si y sólo si el objeto sobre el que se aplica es una instancia del tipo dado. El segundo, `asInstanceOf`, corresponde al operador de conversión de tipos de Java (casting): Si el objeto es una instancia del tipo dado, se ve como tal, en caso contrario se lanza una excepción `ClassCastException`.

Finalmente, el último método por definir es el que comprueba la inferioridad. Hace uso de otro método predefinido, `error`, que lanza una excepción con el mensaje de error dado.

```
  def <(that: Any): boolean = {
    if (!that.isInstanceOf[Date])
      error("cannot compare " + that + " and a Date")

    val o = that.asInstanceOf[Date]
    (year < o.year) ||
```

```
        (year == o.year && (month < o.month ||
                                (month == o.month && day < o.day)))
    }
```

Esto completa la definición de la clase `Date`. Las instancias de esta clase se pueden ver como fechas o como objetos comparables. Es más, todas ellas definen los seis tipos de comparación mencionados antes: `equals` y `<` porque aparecen directamente en la definición de la clase `Date` y los demás porque se heredan del mixin `Ord`.

Por supuesto los mixins son útiles en otras situaciones distintas de la mostrada aquí, pero discutir en profundidad todas sus aplicaciones está fuera del alcance de este documento.

8 Genéricos

La última característica de Scala que vamos a explorar en este tutorial es la generalidad. Los programadores Java veteranos conocen los problemas causados por la falta de la existencia de genéricos en su lenguaje, un defecto corregido en Java 1.5.

La generalidad es la habilidad de escribir código parametrizado por tipos. Por ejemplo, un programador que escriba una librería para listas ordenadas se encuentra con el problema de decidir qué tipo dar a los elementos de la lista. Puesto que esta lista se puede utilizar en contextos muy diversos, no es posible decidir si el tipo de datos de los elementos tiene que ser, por ejemplo, `int`. Eso sería completamente arbitrario y excesivamente restrictivo.

Los programadores Java recurrían al uso de `Object`, que es el super-tipo de todos los objetos. Pero esta solución está lejos de ser ideal puesto que no vale para los tipos básicos(`int`, `long`, `float`, etc.) e hace que el programador tenga que introducir múltiples conversiones dinámicas de tipos.

Scala posibilita definir clases(y métodos) genéricos para solucionar ese problema. Veamoslo con un ejemplo de la clase contendor más sencilla posible: una referencia, que puede ser vacía o apuntar a un objeto de algún tipo.

```
class Reference[a] {
  private var contents: a = _

  def set(value: a) { contents = value }
  def get: a = contents
}
```

Esta clase `Reference` está parametrizada por un tipo de dato, llamado `a`, que es el tipo de sus elementos. Este tipo se utiliza en el cuerpo de la clase como el tipo de la variable `contents`, del argumento del método `set`, y del tipo de respuesta del

método `get`.

El ejemplo anterior de código nos presenta las variables en Scala, que no requieren mayor explicación. Aun así es interesante observar que el valor con el que inicializamos la variable es _, que representa un valor por defecto. Este valor es 0 para los tipos numéricos, **`false`** para el tipo booleano, `()` para el tipo `unit`, y **`null`** para todos los tipos de objetos.

Para utilizar esta clase `Reference`, se ha de especificar que tipo usar para el parámetro a, el que será el tipo del elemento contenido en la referencia. Por ejemplo, para crear y usar un celda que guarde un entero, se podría escribir lo siguiente:

```
object ReferenciaEntera {
  def main(args: Array[String]) {
    val celda = new Reference[Int]
    celda.set(33)
    println("La referencia contiene la mitad de " + (cell.get * 2))
  }
}
```

Como se puede ver en el ejemplo, no es necesario convertir el tipo del valor devuelto por el método `get` antes de usarlo como un entero. Y tampoco se puede almacenar nada que no sea un entero en esa celda particular puesto que ha sido declarada como contenedora de un entero.

9 Conclusión

Este documento da una descripción rápida del lenguaje Scala y muestra algunos ejemplos básicos. El lector interesado puede continuar leyendo el documento *Scala By Example* que contiene ejemplos mucho más avanzados, y consultar la *Especificación del lenguaje Scala (Scala Language Specification)* cuando sea necesario.